Introduction to Linear Algebra using MATLAB
Tutorial on Material Covered in ENG EK 127
Relevant to Linear Algebra

By
Stormy Attaway

Reference: Stormy Attaway, *MATLAB: A Practical Introduction to Programming and Problem Solving*, pp.452+x, Burlington, MA, Elsevier Inc., 2009.

MATLAB Basics

Windows and Prompt

MATLAB can be used in two basic modes. In the Command Window, you can use it interactively; you type a command or expression and get an immediate result. You can also write programs, using scripts and functions (both of which are stored in M-files). This document does *not* describe the programming constructs in MATLAB.

When you get into MATLAB, the configuration may vary depending on settings and the Version number. However, you should have these basic windows:

- the Command Window: this is the main window and is usually on the right
- the Workspace Window: shows current variables
- the Current Directory Window: shows files, by default in the "work" directory (the Current Directory is set using the pull-down menu above the Command Window)
- the Command History Window: shows commands that have been entered in the Command Window

In the Command Window, you should see

```
>>
```

which is the prompt. Any command can be entered at the prompt.

Variables and Assignment Statements

In order to store values, either in an M-file or the Command Window, variables are used. The simplest way to put a value into a variable is to use an assignment statement. An assignment statement takes the form

```
     variable = expression
```

where the name of the variable is on the left-hand side, the expression is on the right-hand side, and the assignment operator is the "=". Variables do not have to be declared in MATLAB. Variables that are defined can be seen in the Workspace Window. In the Command Window, when an assignment statement is entered at the prompt, MATLAB responds by showing the value that was stored in the variable. For example,

```
>> mynum = 5 + 3
mynum =
     8
>>
```

Here, the user entered "mynum = 5 + 3" and MATLAB responded that it had stored in the variable "mynum" the result of the expression, 8.

Putting a semicolon at the end of a statement will suppress the output from that statement, meaning that the command will be carried out, but MATLAB will not show the result. Here is an example of initializing a variable "count" to 0, but suppressing the output, and then adding one to the value of the variable (and not suppressing that output):

```
>> count = 0;
>> count = count + 1
count =
     1
```

MATLAB has a default variable, called “ans”. Any time an expression is entered in the Command Window, without assigning it to a variable, MATLAB will assign the result to the variable “ans”.

```
>> 7-3
ans =
     4
```

There are commands that can be used to see what variables have been created in a given session, and to delete variables. The commands **who** and **whos** will display the current variables (**whos** gives more information). The **clear** command can be used to delete some or all variables.

Constants

There are built-in constants in MATLAB, including:

pi	3.14159….
i, j	$\sqrt{-1}$
inf	infinity ∞
NaN	stands for “not a number”; e.g. the result of 0/0

(Technically, these are built-in functions that return the values shown.)

Operators and Precedence

There are many operators in MATLAB, which can be used in expressions. Some of the arithmetic operators include:

```
+    addition
-    negation, subtraction
*    multiplication
/    division (divided by)
\    division (divided into e.g. 3\12 is 4)
^    exponentiation (e.g. 3^2 is 9)
```

There is a hierarchy, or set of precedence rules, for the operators. For example, for the operators shown here, the precedence is (from highest to lowest):

```
()       parentheses
^        exponentiation
-        negation
*, /, \  all multiplication and division
+, -     addition and subtraction
```

Built-in functions, Help

MATLAB has many built-in functions. It has many mathematical functions, e.g. **abs** for absolute value, **tan** for tangent, etc. The functions are grouped together logically in what are called "help topics". The **help** command can be used in MATLAB to find out what functions are built-in, and how to use them. Just typing "help" at the prompt will show a list of the help topics, the beginning of which is displayed here:

```
>> help

HELP topics:

matlab\general       -  General purpose commands.
matlab\ops           -  Operators and special characters.
matlab\lang          -  Programming language constructs.
matlab\elmat         -  Elementary matrices and matrix manipulation.
matlab\elfun         -  Elementary math functions.
matlab\specfun       -  Specialized math functions.
matlab\matfun        -  Matrix functions - numerical linear algebra.
```

Typing **help** and the name of a help topic (the "matlab\" is not necessary) will show the functions that are contained in that particular grouping. For example, the beginning of the list of operators and special characters is shown here:

```
>> help ops

  Operators and special characters.

  Arithmetic operators.
    plus       - Plus                              +
```

Typing **help** and the name of a function will explain the use of the function.

There are many other functions in Toolboxes with are purchased separately.

Types

There are many built-in types in MATLAB. These are called "classes"; you can see the class of variables when using the **whos** command or in the Workspace window.

By default, the type of numerical expressions and variables is **double**, which is a double precision type for float, or real, numbers. Even when using an integer expression, the default type is **double**. There is also the float type **single**. For integers, there are many built-in types: **int8**, **int16**, **int32**, **int64**; the number in these types represents the number of bits that can be stored in a variable of that type. There are also unsigned integer types, for example, **uint8** is used in true color image matrices.

Other types include **char** for single characters (e.g. 'x') or character strings (e.g. 'hi'), and **logical** for the result of relational, or Boolean, expressions.

All of these type names are also names of functions that can be used to convert a variable or expression to that type. For example:

```
>> clear
>> myvar = 5 + 4;
>> intvar = int32(myvar);
>> whos
  Name         Size                    Bytes  Class

  intvar       1x1                         4  int32 array
  myvar        1x1                         8  double array
```

Vectors and Matrices in MATLAB

MATLAB is written to work with vectors and matrices; the name MATLAB is short for “Matrix Laboratory”. A matrix looks like a table with rows and columns; an *m by n* (or *m x n*) matrix has m rows by n columns (these are the dimensions of the matrix). Vectors are a special case in which one of the dimensions is 1: a row vector is a single row, or in other words it is *1 by n* (1 row by n columns), and a column vector is *m by 1* (m rows by 1 column). A scalar is an even simpler case; it is a *1 by 1* matrix, or in other words, a single value. As seen from the **whos** command, the default in MATLAB is to treat single values as a *1 x 1* matrix.

Creating Vector and Matrix Variables

Row Vectors

Row vector variables can be created in several ways. The simplest method is to put the values that you want in the variable in square brackets, separated by either spaces or commas:

```
>> rowvec = [33 11 15 7]
rowvec =
    33    11    15     7
```

This creates a 1 x 4 row vector “rowvec”, or in other words, a row vector with four elements.

The colon operator can be used to create a row vector that iterates from the starting to ending value with a default step of one; in this case, the square brackets are not necessary:

```
>> itvec = 3:7
itvec =
     3     4     5     6     7
```

A step value can also be specified:

```
>> stepvec = 3:2:11
stepvec =
     3     5     7     9    11
```

Column Vectors

There are two basic methods for creating a column vector: either by putting the values in square brackets, separated by semicolons, or by creating a row vector and then transposing it. The transpose operator in MATLAB is the apostrophe. For example:

```
>> colvec = [33;7;11]
colvec =
    33
     7
    11

>> rowvec = [3:5  44];
>> cvec = rowvec'
cvec =
     3
     4
     5
    44
```

Matrices

Matrix variables can be created by putting the values in square brackets; the values within the rows are separated by either spaces or commas, and the rows are separated by semicolons. **One thing is very important: there must always be the same number of values in every row.** The colon operator can be used to iterate within the rows.

```
>> mat = [1:3; 5 12 0; 6:-1:4]
mat =
     1     2     3
     5    12     0
     6     5     4
```

Special Matrix Functions

There are many built-in functions in MATLAB that create special matrices. Some of them are listed here:

zeros(m,n) creates an m x n matrix of all zeros
zeros(n) creates an n x n matrix of zeros
ones(m,n) creates an m x n matrix of all ones
ones(n) creates an n x n matrix of ones

eye(n) creates an n x n identity matrix (all zeros but ones on the diagonal)
magic(n) creates an n x n magic matrix (sum of all rows, columns, and diagonal are the same)
rand(n) creates an n x n matrix of random real numbers, each in the range from 0 to 1
rand(m,n) creates an m x n matrix of random real numbers, each in the range from 0 to 1

Note: some versions of MATLAB also have a function to create a matrix of random integers, called either **randint** or **randi**. Use **help** to see whether it is available in your version and if so, how to use it.

Referring to Elements

Once you have created a vector or matrix variable, it is necessary to know how to refer to individual elements and to subsets of the vector or matrix. In this section, we will use the following variables to illustrate:

```
>> vec = 4:2:14
vec =
     4     6     8    10    12    14

>> mat = [1:3; 5 12 0; 6:-1:4]
mat =
     1     2     3
     5    12     0
     6     5     4
```

For a vector (row or column), referring to an individual element is done by giving the name of the variable and in parentheses the number of the element (which is called the index or the subscript). In MATLAB, the elements of a vector are numbered from 1 to the length of the vector. For a matrix, both the row and the column index must be given in parentheses, separated by a comma (always the row index and then the column index).

```
>> vec(3)
ans =
     8

>> mat(2,3)
ans =
     0
```

The colon operator can be used to iterate through subscripts. For example, this says the third through fifth elements of the vector (which creates another row vector):

```
>> vec(3:5)
ans =
     8    10    12
```

Using just the colon by itself for a row or column index means all of them. For example, this says all rows within the second column, or in other words, the entire second column:

```
>> mat(:,2)
ans =
     2
    12
     5
```

There is also a special keyword "end" which can be used to refer to the last element in a vector, or the last row and/or column in a matrix (for example, below, the last element in the first row):

```
>> mat(1,end)
ans =
     3
```

These methods of referring to elements can be used to modify elements and in some cases to delete them. To delete element(s), the empty vector [] is assigned. This example creates a vector, modifies a value in it, and then deletes two elements:

```
>> newvec = [22 5 9 11 54];
>> newvec(4) = 49
newvec =
    22     5     9    49    54

>> newvec(2:3) = []
newvec =
    22    49    54
```

Note: this can be done with matrices also, as long as every row in the matrix has the same number of values. This means, for example, that you could not delete an individual element from a matrix, but you could delete an entire row or column.

Dimensions

There are several functions that are used to determine the number of elements in and the dimensions of variables. Assuming a vector variable "vec" and matrix variable "mat",

numel(vec) returns the number of elements in the vector
numel(mat) returns the total number of elements in the matrix (the product of the number of rows and the number of columns)
length(vec) returns the number of elements in the vector
length(mat) returns either the number of rows or the number of columns in the matrix, whichever is larger
size(mat) returns the number of rows and the number of columns of the matrix
size(vec) returns the number of rows and the number of columns of the vector

The **size** function brings up a couple of unique and important concepts in MATLAB:

- A function can return more than one value
- It is possible to have a vector containing more than one value on the left-hand side of an assignment statement. This can be used to store the values that are returned from a function.

For example, to store the number of rows in a matrix in a variable "r" and the number of columns in a variable "c", a vector with the variables "r" and "c" is on the left-hand side of an assignment statement that uses the **size** function to return both of these values:

```
>> mat = [7 11 33 5; 4:7]
mat =
     7    11    33     5
     4     5     6     7

>> [r c] = size(mat)
r =
     2

c =
     4
```

In general, it is NOT good practice to assume the dimensions of a vector or matrix. For a vector variable, typically either **numel** or **length** is used to determine the number of elements in the vector. For a matrix, the assignment statement shown above with **size** is generally used since it is frequently useful to have the number of rows and columns stored in separate variables.

There are also many functions in MATLAB that can change the orientation or dimensions of a matrix:

reshape(mat,m,n) reshapes the matrix as an *m x n* matrix (the number of elements must be the same) by filling the new matrix with the values from the old one column at a time
fliplr(mat) flips the columns from left to right
flipud(mat) flips the rows up and down

Vectors and Matrices as Function Arguments

In MATLAB, an entire vector or matrix can be passed as an argument to a function, and the function will evaluate the function on every element, and return as a result a vector or matrix with the same dimensions as the input argument. For example,

```
>> v = 1:5;
>> exp(v)
ans =
    2.7183    7.3891   20.0855   54.5982  148.4132
```

```
>> mat = [-3 4 11; 0 -2 1]
mat =
    -3     4    11
     0    -2     1
>> abs(mat)
ans =
     3     4    11
     0     2     1
```

For some functions, however, MATLAB will operate column-wise on matrix arguments. For example, there is a function **sum** that will sum the elements in a vector. For a matrix, however, it will sum each individual column. Using the variables “v” and “mat” shown above,

```
>> sum(v)
ans =
    15

>> sum(mat)
ans =
    -3     2    12
```

Matrix Definitions

There are many definitions that relate to matrices, and MATLAB has relevant functions for many of them.

Two matrices are said to be *equal to* each other if they have the same dimensions, and all corresponding elements are equal. In MATLAB, there is a function **isequal** that will receive two matrix arguments and will return logical 1 for true if they are equal, or logical 0 for false if not.

A matrix is *square* if the number of rows is the same as the number of columns. There are definitions that apply only to square matrices.

The *diagonal of a square matrix* is the set of elements from the upper left corner to the lower right; these are the elements for which the row and column indices are the same. The *trace* of a square matrix is the sum of the elements on the diagonal. A *diagonal matrix* is a matrix for which all of the elements that are not on the diagonal are zeros. There are several functions related to square matrices:
trace(squaremat) returns the trace of a square matrix
diag(squaremat) returns the diagonal of a square matrix as a vector
diag(vec) creates a diagonal matrix by creating a matrix of all zeros and putting the vector “vec” on the diagonal

An upper triangular matrix is a square matrix is a matrix that has all zeros below the diagonal. A lower triangular matrix is a square matrix that has all zeros above the diagonal. MATLAB has functions **triu(mat)** and **tril(mat)** that will receive a matrix argument an will create an upper or lower triangular matrix, respectively, by replacing elements with zeros as appropriate.

The diagonal of a matrix is sometimes called the main diagonal. There can be sub-diagonals. For example, a tridiagonal matrix is a matrix that has a main diagonal and a sub-diagonal directly below the main diagonal and another directly above it.

Matrix Operations

Mathematical operations can be performed on matrices; MATLAB has built-in operators and/or functions for them. Since MATLAB is written to work on matrices, loops are not necessary for matrix operation.

For example, *scalar multiplication* means multiplying every element in a matrix by a scalar value. In MATLAB, this is accomplished simply using the "*" operator:

```
>> mat = reshape(1:8,2,4)
mat =
     1     3     5     7
     2     4     6     8

>> mat * 5
ans =
     5    15    25    35
    10    20    30    40
```

Array Operations (term-by-term)

Matrix operations are called "array operations" if they

- operate on two matrices
- the two matrices have the same dimensions
- the operations are performed term-by-term; in other words, on corresponding elements

For array addition and subtraction, the "+" and "-" operators are used. For example, for array addition:

$$\begin{array}{ccccc} A & + & B & = & C \\ \begin{bmatrix} 3 & 4 & 5 \\ 8 & 9 & 10 \end{bmatrix} & + & \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix} & = & \begin{bmatrix} 4 & 7 & 10 \\ 10 & 13 & 16 \end{bmatrix} \end{array}$$

```
>> A = [3:5; 8:10]
A =
     3     4     5
     8     9    10
>> B = reshape(1:6,2,3)
B =
     1     3     5
     2     4     6
```

```
>> C = A + B
C =
     4     7    10
    10    13    16
```

For any operation that is based on multiplication (this means multiplication, division, and exponentiation), the array operator has a dot (".") in front of it. For example, using the matrices A and B above:

```
>> atimesb = A .* B
atimesb =
     3    12    25
    16    36    60

>> araisedtob = A .^ B
araisedtob =
           3          64        3125
          64        6561     1000000
```

Matrix Multiplication

Matrix multiplication is VERY different from the array multiplication defined above. If a matrix A has dimensions *m x n*, then in order to be able to multiply it by a matrix B, B must have dimensions *n x something*; we'll call the column dimension p. In other words, in order to multiply A * B, the number of rows of B has to be the same as the number of columns of A. We say that the "inner dimensions" must agree. The resulting matrix, C, will have as its dimensions *m x p* (the "outer dimensions"). So,

$$[A]_{mxn}\,[B]_{nxp} = [C]_{mxp}$$

This just explains the dimensions; it does not yet describe how the elements in the matrix C are derived. Every element in C is the result of summing the products of corresponding elements in the rows of A and the columns of B. Any given element in C, C_{ij}, is defined as

$$C_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

For example,

$$\begin{array}{ccccc} A & * & B & = & C \\ \begin{bmatrix} 3 & 4 & 5 \\ 8 & 9 & 10 \end{bmatrix} & * & \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} & = & \begin{bmatrix} 26 & 62 \\ 56 & 137 \end{bmatrix} \end{array}$$

The first element in C, C_{11}, is found by multiplying corresponding elements in the first row of A and the first column of B, and summing these, e.g. 3*1 + 4*2 + 5*3 = 26. In MATLAB, matrix multiplication is accomplished using the "*" operator.

```
>> A = [3:5; 8:10]
A =
     3     4     5
     8     9    10

>> B = reshape(1:6,3,2)
B =
     1     4
     2     5
     3     6

>> C = A * B
C =
    26    62
    56   137
```

Note that for square matrices, multiplying a matrix A by an identity matrix I with the same size results in the matrix A (so, multiplying a matrix by I is similar to multiplying a scalar by 1; it doesn't change it).

```
>> A = magic(3)
A =
     8     1     6
     3     5     7
     4     9     2

>> A * eye(3)
ans =
     8     1     6
     3     5     7
     4     9     2
```

Inverse

The definition of the inverse of a matrix is that when multiplying a matrix by its inverse, the result is the identity matrix. Mathematically, we would write this [A] [A^{-1}] = [I]. MATLAB has a built-in function to find an inverse, **inv**.

```
>> A = [1 3; 2 4]
A =
     1     3
     2     4
>> inv(A)
ans =
   -2.0000    1.5000
    1.0000   -0.5000
>> A * inv(A)
ans =
     1     0
     0     1
```

Matrix Augmentation

Matrix augmentation means taking a matrix and augmenting by adding more columns, or another matrix, to it. Note that the dimensions must be correct in order to do this. In MATLAB, this is called concatenating and can be done for either a vector or matrix by putting the two together in square brackets. Here is an example in which a matrix A is augmented with an identity matrix with the same size of A; note that can be obtained with **eye(size(A))**:

```
>> A = [1 3; 2 4]
A =
     1     3
     2     4

>> [A  eye(size(A))]
ans =
     1     3     1     0
     2     4     0     1
```

Vector Operations: Dot Product and Cross Product

As long as the dimensions are correct, some of the definitions and operations given on matrices above are also valid for vectors, since vectors are just a special case of matrices. There are some operations, however, that are only valid for vectors, including the dot product and cross product.

For two vectors A and B that have the same length, the dot product is written A•B and is defined as $\sum_{i=1}^{n} A_i B_i$ = $A_1B_1 + A_2B_2 + A_3B_3 + \ldots + A_nB_n$ where n is the length of the vectors. In MATLAB, there is a function **dot** to accomplish this.

The cross product AxB is defined only if A and B are vectors of length 3, and can be written using the following matrix multiplication:

$$A \text{ x } B = \begin{bmatrix} 0 & -A_3 & A_2 \\ A_3 & 0 & -A_1 \\ -A_2 & A_1 & 0 \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ B_3 \end{bmatrix} = [A_2B_3 - A_3B_2,\ A_3B_1 - A_1B_3,\ A_1B_2 - A_2B_1]$$

MATLAB has a built-in function **cross** for the cross product.

Intro to Linear Algebra: Systems of Equations

A system of linear algebraic equations is of the form

$$\begin{array}{l}
a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n = b_1 \\
a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \ldots + a_{2n}x_n = b_2 \\
a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \ldots + a_{3n}x_n = b_3 \\
\quad\vdots \qquad \vdots \qquad \vdots \qquad\qquad \vdots \qquad \vdots \\
a_{m1}x_1 + a_{m2}x_2 + a_{m3}x_3 + \ldots + a_{mn}x_n = b_m
\end{array}$$

where the a's are the coefficients, the x's are the unknowns, and the b's are constant values.

Using MATLAB, there are two basic methods for solving such a set of equations:

- By putting it in a matrix form
- Using the **solve** function which is part of the Symbolic Math Toolbox

Matrix Form

Because of the way that matrix multiplication works, the system of equations shown above can be written as the product of a matrix of the coefficients A and a column vector of the unknowns x:

$$\begin{array}{ccccc}
\text{A} & \text{x} & = & \text{b} \\
\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{bmatrix} & \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} & = & \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_m \end{bmatrix}
\end{array}$$

We have a matrix multiplication equation of the form Ax = b, and we want to solve for the unknowns x. This can be accomplished as follows:

$$A\,x = b$$

$$A^{-1}\,A\,x = A^{-1}\,b$$

$$I\,x = A^{-1}\,b$$

$$x = A^{-1}\,b$$

So, the solution can be found as a product of the inverse of A and the column vector b.

In MATLAB, there are two ways of doing this, using the built-in **inv** function and matrix multiplication, and also using the "\" operator:

```
>> A = [3 4 1; -2 0 3; 1 2 4]
A =
     3     4     1
    -2     0     3
     1     2     4

>> b = [2 1 0]'
b =
     2
     1
     0

>> x = inv(A) * b
x =
   -1.1818
    1.5000
   -0.4545

>> A\b
ans =
   -1.1818
    1.5000
   -0.4545
```

2 x 2 Systems

The simplest system is a 2 x 2 system, with just two equations and two unknowns. For these systems, there is a simple definition for the inverse of a matrix, which uses the determinant D of the matrix.

For a coefficient matrix A defined generally as

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix},$$

the determinant D is defined as $a_{11}a_{22} - a_{12}a_{21}$.

The inverse $A^{-1} = \frac{1}{D}\begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix}$

For example, for the system

$$\begin{aligned} x_1 + 3x_2 &= -2 \\ 2x_1 + 4x_2 &= 1 \end{aligned}$$

This would be written in matrix form as

$$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -2 \\ 1 \end{bmatrix}$$

The determinant D = 1*4 -3*2 = -2.

The inverse A^{-1} = $\dfrac{1}{-2}\begin{bmatrix} 4 & -3 \\ -2 & 1 \end{bmatrix} = \begin{bmatrix} -2 & 3/2 \\ 1 & -1/2 \end{bmatrix}$

So the solution is: $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -2 & 3/2 \\ 1 & -1/2 \end{bmatrix} \begin{bmatrix} -2 \\ 1 \end{bmatrix} = \begin{bmatrix} 11/2 \\ -5/2 \end{bmatrix}$

MATLAB has a built-in function **det** to find the determinant of a matrix.

```
>> A = [1 3; 2 4]
A =
     1     3
     2     4

>> b = [-2;1]
b =
    -2
     1

>> det(A)
ans =
    -2

>> inv(A)
ans =
   -2.0000    1.5000
    1.0000   -0.5000

>> x = inv(A) * b
x =
    5.5000
   -2.5000
```

Elementary Row Operations

Although finding the inverse of a 2 x 2 matrix is straightforward, as is using it to find the unknowns, it is not so simple for larger systems of equations. Therefore, we resort to other methods of solution. Once the system is in matrix form, some of these methods involve transforming matrices using what are called Elementary Row Operations, or EROs. The important thing to understand about these EROs is that using them to modify a matrix does not change what the solution to the set of equations will be.

There are 3 EROs:

1. Scaling: multiplying a row by a scalar (meaning, multiplying every element in the row by the same scalar). This is written $sr_i \rightarrow r_i$, which indicates that row i is modified by multiplying it by a scalar s.
2. Interchange: interchanging the locations of two rows. This is written as $r_i \leftrightarrow r_j$ which indicates that rows i and j are interchanged.
3. Replacement: replacing all of the elements in one row with that row plus or minus a scalar multiplied by another row. This is written as $r_i \pm sr_j \rightarrow r_i$

These EROs form the basis for the methods described next.

Gauss Elimination

The Gauss Elimination method is a method for solving a matrix equation Ax=b for x. The process is:

1. Start by augmenting the matrix A with the column vector b.
2. Perform EROs to transform this augmented matrix to upper triangular form.
3. Use back-substitution to solve for the unknowns in x.

For example, for a 2 x 2 system, the first step is to augment the matrix [A b]:

$$\begin{bmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \end{bmatrix}$$

Then, EROs are applied to get the augmented matrix into an upper triangular form (which, for a 2 x 2 matrix means finding an ERO that would change a_{21} to 0):

$$\begin{bmatrix} a'_{11} & a'_{12} & b'_1 \\ 0 & a'_{22} & b'_2 \end{bmatrix}$$

Here, the primes indicate that the values may have been changed.

Putting this back into the equation form, we have

$$\begin{bmatrix} a'_{11} & a'_{12} \\ 0 & a'_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b'_1 \\ b'_2 \end{bmatrix}$$

So, we now use the last line which says

$$a'_{22}\ x_2 = b'_2$$

to solve first for x_2:

$$x_2 = b'_2 \ /\ a'_{22}$$

and then go back to the first line which says

$$a'_{11}\ x_1 + a'_{12}\ x_2 = b'_1$$

to solve for x_1:

$$x_1 = (b'_1 - a'_{12}\ x_2)\ /\ a'_{11}$$

Gauss-Jordan Elimination

The Gauss-Jordan Elimination method starts exactly the same way as the Gauss Elimination method, but instead of back-substitution to solve for x, EROs are used to get the augmented matrix into a diagonal form.

1. Start by augmenting the matrix A with the column vector b.
2. Perform EROs to transform this augmented matrix to upper triangular form (forward elimination).
3. Use back elimination (perform more EROs) to get to a diagonal form
4. Solve for the unknowns in x.

For example, for a 3 x 3 matrix, the matrix is first augmented:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{bmatrix}$$

EROs are then performed using forward elimination to get it to upper triangular form:

$$\begin{bmatrix} a'_{11} & a'_{12} & a'_{13} & b'_1 \\ 0 & a'_{22} & a'_{23} & b'_2 \\ 0 & 0 & a'_{33} & b'_3 \end{bmatrix}$$

At this point, the Gauss method would then use back-substitution to solve first for x_3, then x_2, then x_1. Instead, the Gauss-Jordan method continues with back elimination to get it to diagonal form:

$$\begin{bmatrix} a_{11}^{'} & 0 & 0 & b_1^{'} \\ 0 & a_{22}^{'} & 0 & b_2^{'} \\ 0 & 0 & a_{33}^{'} & b_3^{'} \end{bmatrix}$$

Then, the unknowns can be found easily, in any order using these equations:

$$x_i = b_i' / a_{ii}'$$

Reduced Row Echelon Form (RREF)

Reduced Row Echelon Form (RREF) takes the Gauss-Jordan elimination method one step further by performing scaling EROs on all rows so that the a_{ii} coefficients on the diagonal all become ones.

RREF to solve Ax=b for x

To use the RREF method to solve the matrix equation Ax = b for x, the matrix A is augmented with b, and then the Gauss-Jordan method is followed. The final step is to scale all rows. For example, for a 3 x 3 matrix,

$$\begin{bmatrix} a_{11}^{'} & 0 & 0 & b_1^{'} \\ 0 & a_{22}^{'} & 0 & b_2^{'} \\ 0 & 0 & a_{33}^{'} & b_3^{'} \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 0 & b_1^{'} \\ 0 & 1 & 0 & b_2^{'} \\ 0 & 0 & 1 & b_3^{'} \end{bmatrix}$$

Then, the solution is simply the last column. MATLAB has a function **rref** to accomplish this.

RREF to find inverse

This technique and function can also be used to find the inverse of a matrix A. The procedure is:

- Augment A with an identity matrix the same size as A
- Follow the procedure above using EROs to reduce the left-side to an identity matrix; the right side will be the matrix inverse.

For example, for a 3 x 3 matrix, start with [A I]:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & 1 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 1 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & 1 \end{bmatrix}$$

and then reduce this to:

$$\begin{bmatrix} 1 & 0 & 0 & r_{11} & r_{12} & r_{13} \\ 0 & 1 & 0 & r_{21} & r_{22} & r_{23} \\ 0 & 0 & 1 & r_{31} & r_{32} & r_{33} \end{bmatrix}$$ which is [I A^{-1}].

Gauss, Gauss-Jordan, RREF Example

Since the Gauss, Gauss-Jordan and RREF methods to solve Ax = b for x all begin the same way, we will demonstrate all with one example.

The following 3 x 3 system of equations:

$$\begin{aligned} 3x_1 + 2x_2 + x_3 &= 1 \\ x_2 &= 2 \\ x_1 + 2x_2 &= 3 \end{aligned}$$

can be written in matrix form:

$$\begin{bmatrix} 3 & 2 & 1 \\ 0 & 1 & 0 \\ 1 & 2 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

To solve for x, we start with the augmented matrix [A b] and perform forward elimination by finding EROs to get it to upper triangular form:

$$\begin{bmatrix} 3 & 2 & 1 & 1 \\ 0 & 1 & 0 & 2 \\ 1 & 2 & 0 & 3 \end{bmatrix} \quad r_3 - 1/3\, r_1 \rightarrow r_3 \quad \begin{bmatrix} 3 & 2 & 1 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 4/3 & -1/3 & 8/3 \end{bmatrix}$$

$$r_3 - 4/3\, r_2 \rightarrow r_3 \quad \begin{bmatrix} 3 & 2 & 1 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & -1/3 & 0 \end{bmatrix}$$

Now, for the Gauss method, we use back substitution:

$-1/3x_3 = 0$ so $x_3 = 0$

$x_2 = 2$

$3x_1 + 2x_2 + x_3 = 1$
$3x_1 + 2 - 0 = 1$
$3x_1 = -3$
$x_1 = -1$

Instead for Gauss-Jordan, we continue with back elimination:

$$\begin{bmatrix}3 & 2 & 1 & 1\\0 & 1 & 0 & 2\\0 & 0 & -1/3 & 0\end{bmatrix} \; r_1 + 3r_3 \rightarrow r_1 \; \begin{bmatrix}3 & 2 & 0 & 1\\0 & 1 & 0 & 2\\0 & 0 & -1/3 & 0\end{bmatrix} \; r_1 - 2r_2 \rightarrow r_1 \; \begin{bmatrix}3 & 0 & 0 & -3\\0 & 1 & 0 & 2\\0 & 0 & -1 & 0\end{bmatrix}$$

Once in diagonal form, we can solve:

$3x_1 = -3$ so $x_1 = -1$
$x_2 = 2$
$-x_3 = 0$ so $x_3 = 0$

For RREF, scale all rows:

$$\begin{bmatrix}3 & 0 & 0 & -3\\0 & 1 & 0 & 2\\0 & 0 & -1 & 0\end{bmatrix} \; 1/3\, r_1 \rightarrow r_1 \; \begin{bmatrix}1 & 0 & 0 & -1\\0 & 1 & 0 & 2\\0 & 0 & -1 & 0\end{bmatrix} \; -r_3 \rightarrow r_3 \; \begin{bmatrix}1 & 0 & 0 & -1\\0 & 1 & 0 & 2\\0 & 0 & 1 & 0\end{bmatrix}$$

So now the last column is x.

We can also use RREF to find A^{-1} and solve that way.
Start with the augmented matrix [A I]:

$$\begin{bmatrix}3 & 2 & 1 & 1 & 0 & 0\\0 & 1 & 0 & 0 & 1 & 0\\1 & 2 & 0 & 0 & 0 & 1\end{bmatrix} \; r_1 \leftrightarrow r_3 \; \begin{bmatrix}1 & 2 & 0 & 0 & 0 & 1\\0 & 1 & 0 & 0 & 1 & 0\\3 & 2 & 1 & 1 & 0 & 0\end{bmatrix}$$

$$r_3 - 3r_1 \rightarrow r_3 \quad \begin{bmatrix}1 & 2 & 0 & 0 & 0 & 1\\0 & 1 & 0 & 0 & 1 & 0\\0 & -4 & 1 & 1 & 0 & -3\end{bmatrix}$$

$$r_3 + 4r_2 \rightarrow r_3 \quad \begin{bmatrix}1 & 2 & 0 & 0 & 0 & 1\\0 & 1 & 0 & 0 & 1 & 0\\0 & 0 & 1 & 1 & 4 & -3\end{bmatrix}$$

$$r_1 - 2r_2 \rightarrow r_1 \quad \begin{bmatrix}1 & 0 & 0 & 0 & -2 & 1\\0 & 1 & 0 & 0 & 1 & 0\\0 & 0 & 1 & 1 & 4 & -3\end{bmatrix}$$

$$\text{So } A^{-1} = \begin{bmatrix}0 & -2 & 1\\0 & 1 & 0\\1 & 4 & -3\end{bmatrix} \quad \text{and} \quad x = \begin{bmatrix}0 & -2 & 1\\0 & 1 & 0\\1 & 4 & -3\end{bmatrix}\begin{bmatrix}1\\2\\3\end{bmatrix} = \begin{bmatrix}-1\\2\\0\end{bmatrix}$$

In MATLAB, to solve Ax=b we begin by augmenting [A b]:

```
>> A = [3 2 1; 0 1 0; 1 2 0];
>> b = [1:3]';
>> Ab = [A b]
Ab =
     3     2     1     1
     0     1     0     2
     1     2     0     3
```

Performing an ERO in MATLAB can be accomplished by using assignment statements to modify rows. For example this ERO

$$\begin{bmatrix} 3 & 2 & 1 & 1 \\ 0 & 1 & 0 & 2 \\ 1 & 2 & 0 & 3 \end{bmatrix} \quad r_3 - 1/3\, r_1 \rightarrow r_3 \quad \begin{bmatrix} 3 & 2 & 1 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 4/3 & -1/3 & 8/3 \end{bmatrix}$$

is done with:

```
>> Ab(3,:) = Ab(3,:) - 1/3 * Ab(1,:)
Ab =
    3.0000    2.0000    1.0000    1.0000
         0    1.0000         0    2.0000
         0    1.3333   -0.3333    2.6667
```

MATLAB has a function **rref** to reduce:

```
>> rref(Ab)
ans =
     1     0     0    -1
     0     1     0     2
     0     0     1     0
```

In MATLAB, we can find the inverse of a matrix using **rref** and then check that using the **inv** function:

```
>> rref([A eye(size(A))])
ans =
     1     0     0     0    -2     1
     0     1     0     0     1     0
     0     0     1     1     4    -3

>> inv(A)
ans =
    0.0000   -2.0000    1.0000
         0    1.0000         0
    1.0000    4.0000   -3.0000
```

The solve Function in the Symbolic Math Toolbox

If you have the Symbolic Math Toolbox, the **solve** equation can also be used to solve sets of equations. The solution is returned as a structure. Structures are similar to vectors in that they can store multiple values. However, instead of having elements, structures have fields that are named. While indexing is used to refer to elements of a vector, the dot operator is used to refer to the fields in a structure variable.

As an example, we will solve the following 3 x 3 system of equations that was used in the previous section:

$$\begin{aligned} 3x_1 + 2x_2 + x_3 &= 1 \\ x_2 &= 2 \\ x_1 + 2x_2 &= 3 \end{aligned}$$

For simplicity, we will change the x_1,x_2,x_3 to x,y,z, and pass the three equations as strings to the **solve** function:

```
>> result = solve('3*x+2*y+z=1', 'y=2', 'x+2*y=3')
result =
    x: [1x1 sym]
    y: [1x1 sym]
    z: [1x1 sym]
```

This stores the unknowns in fields called x, y, and z within the "result" variable. The dot operator is used to see the actual values, which are stored as symbolic expressions. The **double** function can convert them to numbers:

```
>> result.x
ans =

-1

>> x = double([result.x result.y result.z])'
x =
    -1
     2
     0
```